Detection and Diagnosis of Control Interception

Chang-Hsien Tsai and Shih-Kun Huang

Department of Computer Science, National Chiao Tung University, Taiwan
{chsien,skhuang}@cs.nctu.edu.tw

Abstract. Crash implies that a software is unstable and possibly vulnerable. Stack overflow is one of many causes of crashes. This kind of bug is often hard to debug because of the corrupted stack, so that debuggers cannot trace the control flow of the programs. A control-type crash caused by stack overflow is easy to be developed as a control interception attack. We develop a method to locate this attack and implement it as a plug-in of Valgrind [1]. This tool can be used in the honeypot to detect and diagnose zero-day exploits. We use it to detect several vulnerabilities and automatically locate the bugs.

1 Introduction

According to a recent report [2] developed by the IBM Internet Security Systems, there were 7247 new vulnerabilities in 2006. This number increases nearly 40 percent than the previous year. Over 88 percent of vulnerabilities could be exploited remotely, and over 50 percent allowed attackers to gain access to a machine after exploitation. This serves as a good reason for us to develop a tool to detect attacks and diagnose the vulnerabilities in software.

For better performance, majority of internet servers are implemented in C and C++ programming language. These languages provide low-level operations such as pointer access. While powerful, these operations are the source of common programming errors, one of which is buffer overflow in the stack.

If buffer overflow occurs, the contiguous data are overwritten. This would result in one of three consequences:

1. The program works *normally* like nothing has happened. For example, some data are overwritten, but they will not be used later on. This kind of bug is harmless and may be ignored for a long time.
2. The program works *abnormally*, but most functionalities are as usual. The program may display the extraordinary string or number, because these data are overwritten. This kind of bug is usually easy to identify. With the help of a debugger, the programmer can set *watchpoints* on these suspect variables; the debugger can pause the program as soon as the values of these variables change. There are many researches working on recovering or rollbacking from the abnormal state. Periodic check-points of the system state can be used as replay debugging [3].
3. The program becomes *uncontrollable* because control-sensitive data are corrupted, thereby causing the program to crash.

S. Qing, H. Imai, and G. Wang (Eds.): ICICS 2007, LNCS 4861, pp. 412–426, 2007.

If the flaws can be triggered by user inputs, it is easy for attackers to exploit the bug by intercepting the control flow with carefully crafted input. Even worse, conventional debuggers are handicapped by corrupted stacks . We call this *control interception attack* and develop a tool to detect and diagnose it.

Typical control interception attacks contain *code injection* and *control interception*. Attackers inject malicious code into vulnerable programs. This code is known as shellcode, since the traditional injected code creates a new shell. Through efforts of the hackers [4], new shellcode can even execute as a VNC server.

The second part of the control interception attack is to intercept the control flow of a program. By exploiting the vulnerability of the program, an attacker can overwrite a control-sensitive data to divert the program into the injected code.

1.1 Avoidance of Execution of Injected Code

To mitigate the attacks, many researches attempt to render the injected code harmless. Since the injected code is usually located in the stack or heap, making these areas non-executable [5] would prevent the execution of injected code. However, this technique would cause problems for some software, such as the JIT compiler. Linn et al. [6] observe that successful exploits must invoke system calls. They record the program counter of every invocation of system calls in the executable. The kernel can use the information to differentiate user code from injected code at runtime. Instruction Set Randomization [7][8] *encrypts* trusted binary code with a random key during loading and *decrypts* them during instruction fetching. Injected malicious code becomes garbage and leads to a crash.

Another way is to hinder attackers to predict where the injected code is. Address space layout randomization (ASLR) [5] moves the code segment, stack segment and other segments to different address at each run. PointGuard [9] encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. Without knowing where the injected code is, attackers cannot divert the vulnerable program into these code.

1.2 Detection of Control Corruption

Attackers must corrupt the control-sensitive data to intercept the control flow of the vulnerable program. During every function call in C, there are at least two control-sensitive data in the stack: return address and saved frame pointer.

In this work, we present a method to locate the control-type crash with which conventional debuggers are hard to help. We can report where the control state are corrupted and how the program goes there. The algorithm has been implemented as a plug-in of *Valgrind*[1], called Beagle. We use it to evaluate several vulnerabilities and correctly pinpoint these bugs.

The tool can complement the Valgrind in stack overflow detection. The established *memcheck* plug-in of the Valgrind works perfectly in detecting heap

overflow. However, Valgrind still lacks the capability to detect stack overflow. In the recent Valgrind user survey [10], the stack overflow detection is the most wanted feature. Valgrind developers answer this request in the FAQ 5.2 with

> "Unfortunately, Memcheck doesn't do bounds checking on static or stack arrays. We'd like to, but it's just not possible to do in a reasonable way that fits with how Memcheck works. Sorry."

If the control-sensitive data in the stack are overwritten, our tool can report where the bug is. With this ability, this tool can be a good honeypot to detect zero-day exploit. Because it can detect the attack immediately and does not need to replay the attack. After detecting an attack, it can diagnose the vulnerable code. This tool can also be used with fuzzers (random input generation tools) to find new vulnerabilities.

The remainder of this paper is organized as follows. In Sect. 2, we cover the control-type crash and why it is hard to debug. In Sect. 3, we propose a scheme to point out the bug. In Sect. 4, we detail the implementation of our tool in Valgrind. In Sect. 5, we evaluate our tool with case studies. Finally, in Sect. 6, we present our conclusions.

2 Background

As stated in Sect. 1, uncontrollable programs often lead to crash. Crash implies that there is either an inherent *bug* (programmed by mistake) or a *vulnerability* (triggered by unexpected input). Programs may run out of control and crash due to the corruption of branch control state. Branch control state determines the branch flow of the next instruction for execution, corresponding to three types of branch instructions: function call, function return, and jump. If the branch targets of these instructions are dynamic addresses, they may be corrupted with an invalid address range. For example, dynamic call target can refer to an offset of a virtual table in C++ implementation, or a function pointer in C language. Function returns are usually dynamic. Jump target can also be dynamic. If these targets are corrupted (either unintentionally, or maliciously), the program may fail to meet specifications. Such programs with corrupted control states may also be exploited and thus become vulnerable. It is difficult to reconstruct system failures after a program has crashed due to a corrupted control state and the propagated distance between crash sites and corrupt sites. To cope with this difficulty, we try to monitor running behavior during programs execution. We aim to design a tool that analyzes the program running behavior and determine where the bug is.

First of all, we must clarify that not all crashes are exploitable. We roughly classify two types of crashes: data-type crash and control-type crash. The data-type crash is caused by accessing an illegal memory address; the control-type crash is caused by transferring control to an illegal address. Control-type crash is usually exploitable, while the other one is usually non-exploitable. Secondly, there are two pieces of information that are very helpful in the debugging process:

where the program crashes and how it goes there. Programmers find out the bug more easily with these clues. Normally, it is harder to debug a control-type crash than other crashes, because these clues are missing after a control-type crash.

2.1 Data-Type Crash

Programs often crash resulting from access to illegal memory, which is either an unmapped address or a privileged address. In the Unix environment, programs crash with a message "segmentation fault." To debug a crash, experienced programmers would trace the code from the crash statement rather than the entry point or any other statement, since the bug is usually the crash statement or its preceding statements. Tracing backwards from the crash statement is easier for debugging. If the bug is not in the current function, programmers continue to trace the caller. In this way, it relies on programmer's expertise to find the bug.

It is not easy to identify the crash statement in a big project, if the program does not indicate any message before the crash. A debugger can easily identify the crash statement by reproducing the crash case. For example, the program `crash.c` in Fig. 1 ends with a crash.

```
#include <stdio.h>

void foo(char *p){
    *p = 'x';
}

int main(void){
    char *p;
    p = NULL;
    foo(p);
}
```

Fig. 1. crash.c: sample program with a NULL pointer dereference bug

After running the program in gdb, we get the following messages:

```
Program received signal SIGSEGV, Segmentation fault.
0x0804835e in foo (p=0x0) at crash.c:4
4               *p = 'x';
```

In this case, the program crashes in the function `foo` at the line 4 of `crash.c`, which dereferences a NULL pointer `p`. From the crash statement, we need to trace backwards. The debugger provides a *backtrace*, by which we can follow to trace the caller. We use the command `bt` to print the backtrace as following:

```
(gdb) bt
#0  0x0804835e in foo (p=0x0) at crash.c:4
#1  0x0804838e in main () at crash.c:10
```

This shows that `main()` calls `foo()` ,and the program crashes within `foo()` at line 4. We can search for the bug from the crash statement and then the caller, `main()`, and so on. After finding out the bug, we can easily fix this bug by initializing the pointer `p` with a right value.

Debugging is a backward search; however, conventional debuggers only support forward execution. Programmers need to set breakpoints in the right place and observe the values of variables. If the program crashes before reaching any breakpoint, programmers need to set an earlier breakpoint and restart the process. Thus, programmers must set breakpoints carefully or they will miss the bug. To overcome the difficulty, *Bidirectional debuggers* [11] allow programmers to trace programs forwards as well as backwards.

From this debugging example, we demonstrate that *corrupt statement* is much closer to the real bug than the crash statement. The corrupt statement is where important data are corrupted, thereby causing the crash later on. The corrupt statement is usually the bug itself. If we fix the corrupt statement, the program will not crash. In the aforementioned `crash.c` program, there is a NULL pointer dereference crash. The corrupt statement is '`p = NULL;`' and it is also the bug. To automatically inference from the crash statement to the corrupt statement, Manevich et al. [12] use the static analysis approach.

2.2 Control-Type Crash

After reviewing the debugging process for a data-type crash, we study the control-type crash. The control-type crash is caused by corrupting control-sensitive data, such as the return address in the stack. These control-sensitive data manage the control flow of the program. If one of them get corrupted, the program is out of control when using the corrupted value. The most common corruption is because of the buffer overflow in the stack.

```
void foo(void){
    char buf[8];
    bar(buf);
} /*crash statement*/

void bar(char *buf){
    strcpy(buf, "this is a long string");    /*bug*/
      ...
}
```

Fig. 2. The crash statement and the bug

If a program writes data to a buffer beyond its boundary, other data subsequent to the buffer would be overwritten. The C standard library has many unsafe functions, such as `strcpy()`, `strcat()` and etc. For performance issue, these functions copy data without boundary checking. The buffer is in the stack or heap, and the overflow is referred as stack overflow or heap overflow respectively. In this paper,

we focus on detecting stack overflow. Figure 2 is a sample program with stack overflow vulnerability. The `strcpy()` function writes a long string into the buffer, `buf`, and then the program crashes. As usual, we use gdb to find out the crash statement, but get the following message:

```
Program received signal SIGSEGV, Segmentation fault.
0x7320676e in ?? ()
```

The buggy program crashes, but the gdb cannot report the crash statement in this case. The command `bt` shows no clue as well:

```
(gdb) bt
#0  0x7320676e in ?? ()
#1  0x6e697274 in ?? ()
#2  0xb7fd0067 in ?? () from /lib/tls/libc.so.6
#3  0x080494c4 in ?? ()
#4  0xb7fd7ff4 in ?? () from /lib/tls/libc.so.6
#5  0x00000000 in ?? ()
#6  0xb8000ca0 in ?? () from /lib/ld-linux.so.2
#7  0xbffff358 in ?? ()
#8  0xb7ec6e4b in __libc_start_main() from /lib/tls/libc.so.6
Previous frame inner to this frame (corrupt stack?)
```

We redo the experiment in Microsoft Visual Studio 2003 .NET and get the similar result. Conventional debuggers lose track of the program after the control-type crash, because the control-sensitive data in the stack are corrupted. In this case, programmers must carefully set breakpoints in the debugger before the corruption happens and localize the crash statement in a binary search fashion.

The distinction between the crash statement and the corrupt statement in a control-type crash is essential. The crash statement is obviously where the program crashes, whereas the corrupt statement is where control-sensitive data are corrupted. For example, in Fig. 2, the function *foo* passes its local buffer *buf* to the function *bar*. After calling `strcpy()`, the program's stack is corrupted. However, the program does not crash until the function *foo* returns (in line 4). This example also supports our claim that the corrupt statement is much closer to the bug (the corrupt statement is also the bug).

If attackers overflow the stack with carefully designed values, they can intercept the program. Many kinds of attacks aim to overwrite the control-sensitive data. These attacks contain either a *discrete corruption* or a *continuous corruption*. A discrete corruption is defined as an directly overwrite of control-sensitive data. The typical example of discrete corruption is to overflow via a pointer or a format-string function. A continuous corruption is defined as multiple consecutive writes that overflow the control-sensitive data. The typical example of continuous corruption is the buffer overflow caused by using functions in the C standard library. Attackers can inject any code to execute in the vulnerable program.

3 Detection and Diagnosis of Control Corruption

In this section, we first review several detection methods on control corruption and why they are not precise. Then, we detail our detection method.

3.1 Detection of Library Misuse

Software wrapper is a effective approach to monitor dangerous library call. However it detects only vulnerabilities due to use of library functions. *libsafe* [13] wraps dangerous functions (such as `strcpy()`, `strcat()` and etc.) to enforce boundary checking. Wrapped functions compute the size between the buffer's address and saved frame pointer. If the input data is larger than the size, libsafe halts the program to avoid overwriting the saved frame pointer and the return address. Robertson et al. [14] wrap heap-related functions to detects heap overflow. By wrapping `malloc()`, it inserts canary and padding in front of each memory chunk. By wrapping `free()`, it checksums the chunk to ensure the canary unchanged.

STOBO [15] wraps user functions to detect buffer overflow. It keeps track of lengths of memory buffers and issues warnings when buffer overflows may occur. STOBO finds vulnerabilities in programs even when the test data do not cause overflow, thus sometime issuing false positive.

3.2 Detection of Stack Control Corruption

There are many researches about detecting stack overflow. When they detect corruption in the control-sensitive data, they will terminate the process to avoid executing malicious code. There are two approaches to detect corruption: *canary* and *backup*.

Canary approach is used by StackGuard [16]. A canary is a special value inserted before the saved return address when a new stack frame is allocated. Any attempt to overwrite the saved return address will also overwrite the canary. Just before the function returns, the canary will be checked. If the canary changes, StackGuard detects a stack overflow and terminates the process. The StackGuard aims to protect the saved return address, but it leaves another control-sensitive data, the save framed pointer, under attack. *SSP* [17] and Microsoft Visual Studio /GS option [18] enhance the method by inserting the canary between the saved frame pointer and local variables. The canary approach works fine for continuous corruption. Nevertheless, it may not detect the discrete corruption since the canary can be bypassed.

Backup approach is used by StackShield [19]. It backs up the return address of the current function in another global variable when a new stack frame is allocated. When the function returns, it compares the return address with the stored one. If the value changes, StackShield detects a stack overflow and terminates the process. The same approach is implemented in Win32 PE binary programs [20] as well as DLL [21]. VtPath [22] extracts return addresses from the call stack in a training phase and uses them to detect exploits in runtime.

These approaches all neglect the saved frame pointer. Our tool takes the backup approach since it is better than the canary approach, and we back up both the saved return address and the saved frame pointer.

Other than the aforementioned shortcomings, these methods are designed to detect attacks and kill itself to avoid executing the injected code. Hence, they can not find the corrupt site. The first reason is that they check at the epilog of every function. Therefore, any corruption occurred in a function is not detected until the epilog of the function. The larger the function is, the more imprecise the method is. Programmers must waste much time in finding the corrupt statement backwards in the function. Due to this reason, our method's detection granularity is a basic block. If the corrupt statement is in the library, the bug is usually in the line of function invocation, because most faults occur in misuse of the library, not the implementation of the library itself, for example, the use of `strcpy()`.

After enhancing the detection timing, there is still another problem. If one function corrupts its caller's control-sensitive data via the pointer, the corruption is not detected until returning to the caller. For example in Fig. 2, the function `foo` passes a pointer of its local variable `buf` to the function `bar`, which calls `strcpy()` to overwrite `foo`'s local variable and return address. The program's control-sensitive data are corrupt, but the program has not yet crashed until the function `foo` returns[1]. To detect this corruption as soon as possible, we need to check every control-sensitive data in the stack, rather than those of current function only. In this way, we can detect the function `foo`'s control-sensitive data are corrupted after the function `strcpy` is called.

3.3 Localization of the Corrupt Statement

We first instrument a *Backup* function in every function's prolog and instrument a *Verify* function in the end of every basic block. If the Verify function find a mismatch, it will report the corrupt statement and the stack trace. It also reports which stack frame is corrupt (victim frame). With this detection mechanism we have to ensure that these functions will not disturb the program's normal execution. The jobs of these functions are:

Backup saves the current frame's saved frame pointer and return address. The Backup function also needs to track which frame we are executing. We will cover this in the next section.

Verify compares current frame's saved frame pointer and return address with backups. Then it compares the previous frame's frame pointer and return address with corresponding backups and so on until the `main` function. If one of these does not match the backup values, the *Report* function will be called.

Report will report the victim stack frame and the backtrace. The currently executing statement is the corrupt statement, and the first unmatched frame

[1] Function `foo`'s return address and saved frame pointer are corrupt, but function `bar`'s are normal.

is the victim frame[2]. It is easy to reconstruct the backtrace since we have all the return addresses before corruption. We can infer the calling addresses from these return addresses.

4 Implementation in Valgrind

We implement this method as a plug-in of Valgrind, which is a JIT-based emulator for linux. At runtime, each basic block in the binary is translated into a RISC-like assembly language, called UCode, and the instruments it with Backup and Verify function. The instrumented block is then translated back into the native code to execute.

In Bealge plug-in, we first find the base address of `main`'s stack frame. The data whose address is beyond the base address are used by the startup code of libc, and it is not our concern. The base address will be used as the boundary in the Verify function. When a new basic block comes to the plug-in, we map the code address to a symbol name by using `VG_(get_fnname_if_entry)` function. If the symbol name is `"main"`, we are instrumenting the `main` function. We save the frame pointer of this function as `main_ebp`, which is the boundary.

After recoding the `main_ebp`, we need to instrument Backup function in the prolog of every function. Normally, a function prolog is like:

```
pushl %ebp
movl %esp, %ebp
```

The `movl %esp, %ebp` is translated to UCode as:

```
GETL %ESP, t6
PUTL t6, %EBP
INCEIPo $2
```

We check each instruction in a basic block. If the opcode is "`PUT`" and the value is `EBP`, the program is modifying the `EBP` register. This usually indicates that a new stack frame is allocated. We instrument the Backup function after such instruction.

We instrument the Verify function before the last instruction of every basic block. In this way, we can make sure the corrupt statement is in this block, if there is a mismatch found in the Verify function. If so, we collect all the return addresses as an array for the parameter of `VG_(mini_stack_dump)`, which prints the stack trace.

A simple implementation of the Verify function is a stack walk algorithm. Figure 3(a) shows the normal stack where every saved frame pointer points to the

[2] In the most cases, if the frame pointer in the victim frame does not match the backup, this indicates a continuous corruption. The victim buffer is allocated as this function's local variable. If the return address does not match the backup, but the frame pointer matches the backup, it indicates a discrete corruption on the return address.

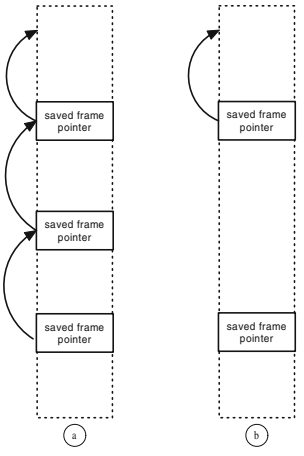

Fig. 3. (a) The normal stack frame (b) Frame pointer omission

previous saved frame pointer. In the algorithm, the saved frame pointer is first compared with the backup. If the values match, it is used as the address to find the saved frame pointer of the parent function. The saved frame pointer of the parent function is then compared with the corresponding backup. The stack walk algorithm goes on until reaching the `main` function (comparing with the `main_ebp`).

4.1 Frame Pointer Omission

The simple stack walk implementation works fine for normal case; however, a compiler optimization technique, called *Frame Pointer Omission* (FPO), may complicate the task. As there are only *seven* general registers in the x86 CPU it is undesirable to dedicate one of them, the `EBP` register, for addressing local variables. Contemporary compilers can generate code that addresses local variables via the `ESP` register instead of the `EBP` register. In gcc, the FPO feature is enabled by the `-fomit-frame-pointer` option, which is implied by several optimization levels.

With FPO, some functions may have no formal stack frame. The `EBP` registers during these functions serve for general purpose rather than pointing to the saved frame pointer. As shown in Fig. 3(b), the static link used by the saved frame pointer is "broken" in these functions. Considering the FPO, we must not only store the value of the save frame pointer but also the value of the `EBP` register.

In the implementation, an array `ebp` stores all the previous seen values of the EBP register. An array `savefp` stores all the previous saved frame pointers. An array `retaddr` stores all the return addresses. The index variable `nframe` is tracking which frame we are executing. In the Backup function, there are several cases:

***EBP == ebp[nframe]** This is a new stack frame, and the variable `nframe` is increased by one. The saved frame pointer, the value of the EBP register and the return address are backed up.

EBP == ebp[nframe-1] We returns to the parent function, and the variable `nframe` is decreased by one.
***EBP == savefp[nframe]** We are in the same function.

4.2 `setjmp` and `longjmp` Functions

The C standard library provides two functions, `setjmp` and `longjmp`, to transfer control directly from one function to another currently executing function (one of the parent functions) without going through the normal return sequence. The `setjmp` function saves the current stack context for later use by `longjmp`. There may be several `longjmp` calls, each of which represents one case of exception. After the `longjmp` is called, the control will directly transfer to the recently called `setjmp` with different return values to distinguish the exceptions.

In the implementation of Verify function, we must consider the presence of the `longjmp` call. In the normal case, the value of the `EBP` register will be the last used element of the `ebp` array (indexed by `nframe`). However, after the `longjmp`, the `EBP` register will point to the previous frame where the `setjmp` is called. The Verify function must first find the matched value of the `EBP` register in the `ebp` array before starting matching the `retaddr` and `savefp` array.

5 Evaluations

To validate the correctness of our tool, we need to verify that our tool does point out real bugs. In this section, we check several vulnerabilities in open source projects.

5.1 Picasm Error Handling Stack Overflow Vulnerability

Picasm is a Microchip PIC16Cxx assembler, designed to run on most UNIX-like operating systems. When generating error and warning messages, picasm copies strings into fixed length buffers without bounds checking. Below is one of the vulnerable functions.

```
152 void
153 error(int lskip, char *fmt, ...)
154 {
155   va_list args;
156   char outbuf[128];
157
158   err_line_ref();
159   strcpy(outbuf, "Error: ");
160   va_start(args, fmt);
161   vsprintf(outbuf+7, fmt, args);
```

We run the crash case under Beagle and get the following report:

```
corrupted in frame[1]
==10941==    at 0x3AA00D6C: mempcpy (mempcpy.S:58)
==10941==    by 0x3A9D2AD8: vfprintf (vfprintf.c:1535)
==10941==    by 0x3A9EBD2A: vsprintf (iovsprintf.c:46)
==10941==    by 0x8048FBD: error (picasm.c:161)
==10941==    by 0x8049F3A: main (picasm.c:887)
```

The first line indicates the victim frame is the first frame, function `error` , because its local variable `outbuf` is overflow. The next lines indicate the backtrace at the corrupt statement. The "==10941==" is the process id of the process. The corrupt statement is the `vsprintf` in line 161 of `picasm.c`, which copies the error message into the `outbuf` without boundary checking.

5.2 Buffer Overflow in Elm (Expires Header)

Elm is a popular mail user agent for Unix. There is a stack overflow in processing mail header disclosed in Aug 2005. Attackers can craft the following mail with overlong `Expires` header to crash the elm:

```
From: attacker@localhost
To: user@victim.com
Subject: Elm buffer overflow
Expires: UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

The vulnerability exists in `expires.c:process_expiration_date` that processes the `Expires` header:

```
41 process_expiration_date(date, message_status)
42 char *date;
43 int *message_status;
44 {
   ....
72     items = sscanf(date, "%s %s %s %s %s",
73                 word1, word2, word3, word4, word5);
```

The sscanf() in line 72 reads the string into local variable `word1`, `word2`, `word3`, `word4` and `word5`. The size of these variable is 20 bytes. It enough for date, but is far less than arbitrary string as the crafted email. We use Bealge to get the following result:

```
corrupted in frame[3]
==13073== at 0x3AA3794B: _IO_sputbackc (in /lib/tls/libc-2.3.4.so)
==13073== by 0x3AA214F7: _IO_vfscanf (in /lib/tls/libc-2.3.4.so)
==13073== by 0x3AA2D5A8: vsscanf (in /lib/tls/libc-2.3.4.so)
==13073== by 0x3AA2884A: sscanf (in /lib/tls/libc-2.3.4.so)
```

```
==13073== by 0x8058BAC: process_expiration_date (expires.c:72)
==13073== by 0x8068B07: read_headers (newmbox.c:705)
==13073== by 0x806796C: newmbox (newmbox.c:184)
==13073== by 0x8055B2D: main (elm.c:108)
```

The Elm starts from `main` and then `newmbox`, and so on. The third frame, `process_expiration_date`, is corrupted by the `sscanf` in line 72 of the file `expires.c`.

5.3 Berlios GPSd gpsd_report() Format String Vulnerability

Berlios GPSd is a daemon that monitors GPSes attached to a computer and makes all data available at a TCP socket. In GPSd versions 1.9.0 through 2.7, there is a format string vulnerability that the file `gpsd_report` calls `syslog` with an input from user.

```
112         syslog((errlevel == 0) ? LOG_ERR : LOG_NOTICE, buf);
```

We attack this server with the exploit from Metasploit [4] and find out the bug as following.

```
==15277== Process terminating with default action of signal 11 (SIGSEGV)
==15277==  Access not within mapped region at address 0x3A746E65
==15277==    at 0x3A9E2032: vfprintf (in /lib/i686/libc-2.4.so)
==15277==    by 0x3AA6AAF5: __vsyslog_chk (in /lib/i686/libc-2.4.so)
==15277==    by 0x3AA6ACE9: syslog (in /lib/i686/libc-2.4.so)
==15277==    by 0x8049077: gpsd_report (gpsd.c:112)
==15277==    by 0x804A633: main (gpsd.c:620)
```

5.4 Comparison with CRED

In our survey, CRED [23] is the most related work to ours and its source code is available, so we compare it with our work. From the standpoint of program language, any pointer access out of its storage is a bug. However, there is no size information in a pointer in C language. Jones and Kelly [24] store pointer address and size information for run-time checks in a splay tree. CRED is an extension of Jones and Kelly's work to allow OOB access.

Both as dynamic analysis, CRED is designed to detect buffer overflow, whereas Beagle is designed to detect control corruption. CRED can detect buffer overflow in the stack, but we can not detect some cases if the quasi-invariant is not violated. Nevertheless, CRED can not detect the three vulnerabilities presented in this section, because it does not handle the format-string functions. In addition, CRED can not detect the overflow caused by system call, such as `read()`. Valgrind can not instrument the system call as well, but it will detect the corruption after the system call. This is another advantage to use our method.

As shown in Table 5.4, we conduct several experiments to compare the performance on a 3.4Ghz Intel Pentium 4, Linux system using gcc 4.0.2. Gzip and bzip2

Table 1. Performance of Analysis (seconds)

	gzip-1.3.12	ccrypt-1.7	bzip2-1.0.4
gcc	0.02	0.01	0.02
CRED	0.20	0.79	0.24
beagle	0.30	0.25	0.30

are used to decompress their tarball, and ccrypt is used to encrypt a file. Both CRED and beagle suffer from great performance loss compared with original program. Generally speaking, Beagle runs slower than CRED. However, CRED has worst performance in ccrypt, which has many pointer operations.

6 Conclusion

Unreliable software with inherent bugs may be exploited to violate security specifications, meant to be security faults. We design and implement a tool to backtrack the control-type crash. We can detect control corruption caused by stack overflow, format-string attacks or directly overwrites. It can be an effective tool to diagnose the control-type crash. It is also a good tool to detect and analyze security attacks.

Acknowledgement

This work was sponsored in part by NSC project NSC96-3114-P-001-002-Y, NSC95-2221-E-009-068-MY2, NSC 96-2221-E-001-026, and TWISC@NCTU.

References

1. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: Sokolsky, O., Viswanathan, M. (eds.) Electronic Notes in Theoretical Computer Science, vol. 89, Elsevier, Amsterdam (2003)
2. IBM Internet Security Systems: Ibm report: Software security vulnerabilities will continue to rise in 2007,(2007) `http://www.iss.net/about/press_center/releases/us_ibm_report.html`
3. Srinivasan, S.M., Kandula, S., Andrews, C.R., Zhou, Y.: Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In: USENIX Annual Technical Conference, General Track, pp. 29–44 (2004)
4. Metasploit Team: Metasploit project, `http://www.metasploit.com/`
5. PaX Team: Documentation for the pax project, `http://pax.grsecurity.net/docs/index.html`
6. Linn, C.M., Rajagopalan, M., Baker, S., Collberg, C., Hartman, J.H.: Protecting against unexpected system calls. In: Proceedings of the 2005 USENIX Security Symposium, pp. 239–254 (2005)

7. Barrantes, E.G., Ackley, D.H., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: CCS 2003: Proceedings of the 10th ACM conference on Computer and communications security, pp. 281–289. ACM Press, New York (2003)
8. Sovarel, A.N., Evans, D., Paul, N.: Where's the feeb?: The effectiveness of instruction set randomization. In: Proceedings of 14th USENIX Security Symposium (2005)
9. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium, USENIX, pp. 91–104 (2003)
10. Valgrind Team: 2nd official valgrind survey (2005), `http://valgrind.org/gallery/survey_05/summary.txt`
11. Biswas, B., Mall, R.: Reverse execution of programs. ACM SIGPLAN Notices 34(4), 61–69 (1999)
12. Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: Pse: explaining program failures via postmortem static analysis. In: SIGSOFT 2004/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pp. 63–72. ACM Press, New York (2004)
13. Baratloo, A., Tsai, T., Singh, N.: Libsafe: Protecting critical elements of stacks. White paper, Bell Labs, Lucent Technologies (1999)
14. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: proceedings of 17th USENIX Large Installation Systems Administration (LISA) Conference (2003)
15. Haugh, E., Bishop, M.: Testing c programs for buffer overflow vulnerabilities. In: Proceedings of the 2003 Symposium on Networked and Distributed System Security (2003)
16. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. 7th USENIX Security Conference, San Antonio, Texas, pp. 63–78 (1998)
17. Etoh, H.: Gcc extension for protecting applications from stack-smashing attacks, `http://www.trl.ibm.com/projects/security/ssp/`
18. Bray, B.: Compiler security checks in depth. Technical report, Microsoft Corporation (2002)
19. Vendicator: Stack shield: a "stack smashing" technique protection tool for linux,(2000) `http://www.angelfire.com/sk/stackshield/`
20. Prasad, M., cker Chiueh, T.: A binary rewriting defense against stack based overflow attacks. In: Proceedings of the USENIX Annual Technical Conference, pp. 211–224 (2003)
21. Nebenzahl, D., Sagiv, M.: Install-time vaccination of windows executables to defend against stack smashing attacks. IEEE Trans. Dependable Secur. Comput. 3(1), 78 (2006) (Senior Member-Avishai Wool)
22. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings of the 2003 Symposium on Security and Privacy, pp. 62–77. IEEE Computer Society Press, Los Alamitos (2003)
23. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the 11th Annual Network and Distributed System Security Symposium (2004)
24. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: AADEBUG, pp. 13–26 (1997)